

Control Flow Analysis for SF Combinator Calculus

Martin Lester

Department of Computer Science, University of Oxford, Oxford, UK

`martin.lester@cs.ox.ac.uk`

Programs that transform other programs often require access to the internal structure of the program to be transformed. This is at odds with the usual extensional view of functional programming, as embodied by the lambda calculus and SK combinator calculus. The recently-developed SF combinator calculus offers an alternative, intensional model of computation that may serve as a foundation for developing principled languages in which to express intensional computation, including program transformation. Until now there have been no static analyses for reasoning about or verifying programs written in SF-calculus. We take the first step towards remedying this by developing a formulation of the popular control flow analysis OCFA for SK-calculus and extending it to support SF-calculus. We prove its correctness and demonstrate that the analysis is invariant under the usual translation from SK-calculus into SF-calculus.

Keywords: control flow analysis; SF-calculus; static analysis; intensional metaprogramming

1 Introduction

In order to reason formally about the behaviour of program transformations, we must simultaneously consider the semantics of both the program being transformed and the program performing the transformation. In most languages, program code is not a first-class citizen: typically, the code and its manipulation and execution are encoded in an ad-hoc manner using the standard datatypes of the language. Consequently, whenever we want to reason formally about program transformation, we must first formalise the link between the encoding of the code and its semantics.

This is unsatisfying and it would be desirable to develop better techniques for reasoning about program transformation in a more general way. In order to do this, we must develop techniques for reasoning about programs that manipulate other programs. That is, we need to develop techniques for reasoning about and verifying uses of *metaprogramming*. Metaprogramming can be split into *extensional* and *intensional* uses: extensional metaprogramming involves joining together pieces of program code, treating the code as a “black box”; intensional metaprogramming allows inspection and manipulation of the internal structure of code values.

Unfortunately, as support for metaprogramming is relatively poor in most programming languages, its study and verification is often not a priority. In particular, the λ -calculus, which is often thought of as the theoretical foundation of functional programming languages, does not allow one to express programs that can distinguish between two extensionally equal expressions with different implementations, or indeed to manipulate the internal structure of expressions in any way.

However, the SF combinatory calculus [6] does allow one to express such programs. SF-calculus is a formalism similar to the familiar SK combinatory calculus, which is itself similar to λ -calculus, but avoids the use of variables and hence the complications of substitution and renaming. SF-calculus replaces the K of SK-calculus with a factorisation combinator F that allows one to deconstruct or *factorise* program terms in certain normal forms. Thus it may be a suitable theoretical foundation for programming languages that support intensional metaprogramming.

There has been some recent work on verification of programs using extensional metaprogramming, mainly in languages that only allow the composition and execution of well-formed code templates [1, 4]. In contrast, verification of intensional metaprogramming has been comparatively neglected.

There do not yet appear to be any *static analyses* for verifying properties of SF-calculus programs. We rectify this by formulating the popular analysis *OCFA* [20] for SF-calculus. We prove its correctness and argue, with reference to a new formulation of OCFA for SK-calculus, why it is appropriate to call the analysis OCFA. This provides the groundwork for more expressive analyses of programs that manipulate programs.

We begin in Section 2 by reviewing SK-calculus and OCFA for λ -calculus; we also present a summary of SF-calculus. In Section 3, we reformulate OCFA for SK-calculus and prove its correctness; this guides our formulation and proof of OCFA for SF-calculus in Section 4. We discuss the precision of our analysis in Section 5 and compare it with some related work in Section 6. We conclude by suggesting some future research directions in Section 7.

2 Preliminaries

2.1 OCFA for Lambda Calculus

OCFA [20] is a popular form of *Control Flow Analysis*. It is flow insensitive and context insensitive, but precise enough to be useful for many applications, including guiding compiler optimisations [17, 2], providing autocompletion hints in IDEs and noninterference analysis [14]. It is perhaps the simplest static analysis that handles higher order functions, which are a staple of functional programming.

Let us consider OCFA for the λ -calculus. OCFA can be formulated in many ways. Following Nielson and others, we present it as a system of constraints [18]. Suppose we wish to analyse a program e . We begin by assigning a unique label l (drawn from a set *Label*) to every subexpression (variable, application or λ -abstraction) in e . (Reusing labels does not invalidate the analysis, and indeed this is done deliberately in proving its correctness, but it does reduce its precision.) We write e^l to make explicit reference to the label l on e . We often write applications infix as $e_1 @^l e_2$ rather than $(e_1 e_2)^l$ to make reference to their labels clearer. We follow the usual convention that application associates to the left, so $f g x$ (or $f @ g @ x$) means $(f g) x$ and not $f (g x)$.

Next, we generate constraints on a function Γ by recursing over the structure of e , applying the rules shown in Figure 1. Finally, we solve the constraints to produce $\Gamma : \text{Label} \uplus \text{Var} \rightarrow \mathcal{P}(\text{Abs})$ that indicates, for each position indicated by a subexpression label l or variable x , an over-approximation of all possible expressions that may occur in that position during evaluation. Abstractly represented values v have the form $FUN(x, l)$, indicating any expression $\lambda x. e^l$ that binds the variable x to a body with label l . We say that $\Gamma \models e$ if Γ is a solution for the constraints generated over e .

The intuition behind the rules for $\Gamma \models e$ is as follows:

- If $e = x^l$: $\Gamma(x)$ must over-approximate the values that can be bound to x .
- If $e = \lambda^{l_1} x. e^{l_2}$: A λ -expression is represented abstractly as $FUN(x, l_2)$ by the variable it binds and the label on its body. Furthermore, its subexpressions must be analysed.
- If $e = e_1^{l_1} @^l e_2^{l_2}$: For any application, consider all the functions that may occur on the left and all the arguments that may occur on the right. Each argument may be bound to any of the variables in the functions. The result of the application may be the result of any of the function bodies. Again, all subexpressions of the application must be analysed.

Labels	$Label \ni l$
Variables	$Var \ni x$
Labelled Expressions	$e ::= x^l \mid e_1 @^l e_2 \mid \lambda^l x. e$
Abstract Values	$Abs \ni v ::= FUN(x, l)$
Abstract Environment	$\Gamma : Label \uplus Var \rightarrow \mathcal{P}(Abs)$

$\Gamma \models x^l$	\iff	$\Gamma(x) \subseteq \Gamma(l)$
$\Gamma \models \lambda^l x. e^l$	\iff	$\Gamma \models e^l \wedge FUN(x, l_2) \in \Gamma(l_1)$
$\Gamma \models e_1^l @^l e_2^l$	\iff	$\Gamma \models e_1^l \wedge \Gamma \models e_2^l \wedge (\forall FUN(x, l_3) \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(x) \wedge \Gamma(l_3) \subseteq \Gamma(l))$

Figure 1: OCFA for λ -calculus

In order to argue about the soundness of the analysis, we must first formalise what Γ means. We can do this via a *labelled semantics* for λ -calculus that extends the usual rules for evaluating λ -calculus expressions to labelled expressions. We can then prove a coherence theorem [24]: if $\Gamma \models e^l$ and (in the labelled semantics) $e^l \rightarrow e'^l$, then $\Gamma \models e'^l$ and $\Gamma(l') \subseteq \Gamma(l)$. In fact, by induction on the length of derivations of \rightarrow^* , this is also true for $e^l \rightarrow^* e'^l$. Note in particular that, as $\Gamma(l') \subseteq \Gamma(l)$, $\Gamma(l)$ gives a sound over-approximation to the subexpressions that may occur at the top level at any point during evaluation.

As a concrete example, consider the λ -expression $(\lambda^1 x. x^0) @^4 (\lambda^3 y. y^2)$, which applies the identity function to itself. We have chosen *Label* to be the natural numbers \mathbb{N} . A solution for Γ is:

$$\Gamma(x) = \Gamma(0) = \Gamma(3) = \Gamma(4) = \{FUN(y, 2)\} \quad \Gamma(1) = \{FUN(x, 0)\} \quad \Gamma(y) = \{\}$$

In particular, this correctly tells us that the result of evaluating the expression is abstracted by $FUN(y, 2)$; that is, the identity function with body labelled 2 that binds y .

Note that the constraints on Γ may easily be solved by: initialising every $\Gamma(l)$ to be empty; iteratively considering each unsatisfied constraint in turn and enlarging some $\Gamma(l)$ in order to satisfy it; stopping when a fixed point is reached and all constraints are satisfied. Done naively, this takes time $\mathcal{O}(n^5)$ for a program of size n [18]. With careful ordering of the consideration of constraints, this improves to $\mathcal{O}(n^3)$. The best known algorithm for OCFA uses an efficient representation of the sets in Γ to achieve $\mathcal{O}(n^3 / \log n)$ complexity [3]. Van Horn and Mairson showed that, for linear programs (in which each bound variable occurs exactly once), OCFA gives the same result as actually evaluating the program; hence it is PTIME-complete [10].

OCFA has been the inspiration for many other analyses. For example, k -CFA adds k levels of context to distinguish between uses of the same function from different points within a program. This improves precision, but at the cost of making the analysis EXPTIME-complete, even for $k = 1$ [9]. CFA2 similarly tries to use context to improve precision, but via a pushdown abstraction, which remains practical [23].

2.2 SK Combinatory Calculus

Combinatory logic is a Turing-powerful formalism for computation that is similar in style to the λ -calculus, but without bound variables and the associated complications of capture-avoiding substitution and α -conversion [8]. From the perspective of term rewriting systems, a combinator is a named constant

C with an associated rewrite rule $C \bar{x} \rightarrow t(\bar{x})$, where \bar{x} is a sequence of variables of fixed length and $t(\bar{x})$ is term built from the variables in \bar{x} using application; that is, $t(\bar{x})$ is an applicative term.

The *SK Combinatory Calculus* (or just SK-calculus) is the rewrite system involving terms built from just two atomic combinators, S and K :

$$\begin{aligned} S f g x &\rightarrow f x (g x) \\ K x y &\rightarrow x \end{aligned}$$

A combinator can also be viewed as a function acting on terms; hence applicative terms t built from combinators are also functions. Using just S and K , it is possible to express all functions encodable in the λ -calculus. For example, $S K K$ encodes the identity function. Figure 2 shows the rewrite rules and the evaluation of the identity with terms depicted as trees.

From a λ -calculus perspective, a combinator can be viewed as a closed λ -term built by wrapping a purely applicative term in λ -abstractions:

$$\begin{aligned} S &\equiv \lambda f. \lambda g. \lambda x. f x (g x) \\ K &\equiv \lambda x. \lambda y. x \end{aligned}$$

This leads to an obvious translation $lambda(t)$ from SK-calculus into λ -calculus:

$$\begin{aligned} lambda(S) &\stackrel{def}{=} \lambda f. \lambda g. \lambda x. f x (g x) \\ lambda(K) &\stackrel{def}{=} \lambda x. \lambda y. x \\ lambda(t_1 t_2) &\stackrel{def}{=} lambda(t_1) lambda(t_2) \end{aligned}$$

There are a number of translations $unlambda(e)$ from λ -calculus into SK-calculus, including the following [8]:

$$\begin{aligned} unlambda(x) &= x \\ unlambda(e_1 e_2) &= unlambda(e_1) unlambda(e_2) \\ unlambda(\lambda x. e) &= unlambda_x(e) \\ unlambda_x(x) &= S K K \\ unlambda_x(e) &= K unlambda(e) && \text{if } x \text{ does not occur free in } e \\ unlambda_x(e x) &= unlambda(e) && \text{if } x \text{ does not occur free in } e \\ unlambda_x(e_1 e_2) &= S unlambda_x(e_1) unlambda_x(e_2) && \text{if neither of the above applies} \\ unlambda_x(\lambda y. e) &= unlambda_x(unlambda(\lambda y. e)) \end{aligned}$$

This translation is left-inverse to the λ -translation; that is $unlambda(lambda(t)) = t$. However, it is not right-inverse.

The rewrite rules of combinatory calculus are very simple to implement, as: there is no need to track bound variables; the number of rewrite rules is small and fixed; and all transformations are *local*. Here “local” means that, viewing a term as a graph, each transformation involves only a small, bounded number of edge additions and deletions, all affecting nodes that are either within a bounded distance of the combinator or are newly created (with the number of new nodes also being bounded). Because of this simplicity, combinators have frequently been considered as a basis for hardware or virtual machines for executing functional programs [22, 5]. Combinators can be thought of as an assembly language for functional programs (although often an expanded set of combinators [21] is used to avoid a combinatorial explosion in the size of the compiled program).

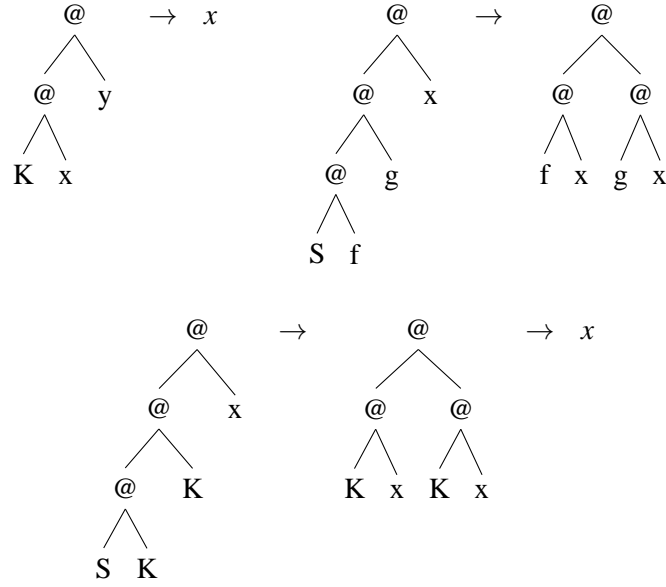


Figure 2: Terms of SK-calculus viewed as trees. Above: the reduction rules for S and K . Below: evaluation of the identity function $S K K$.

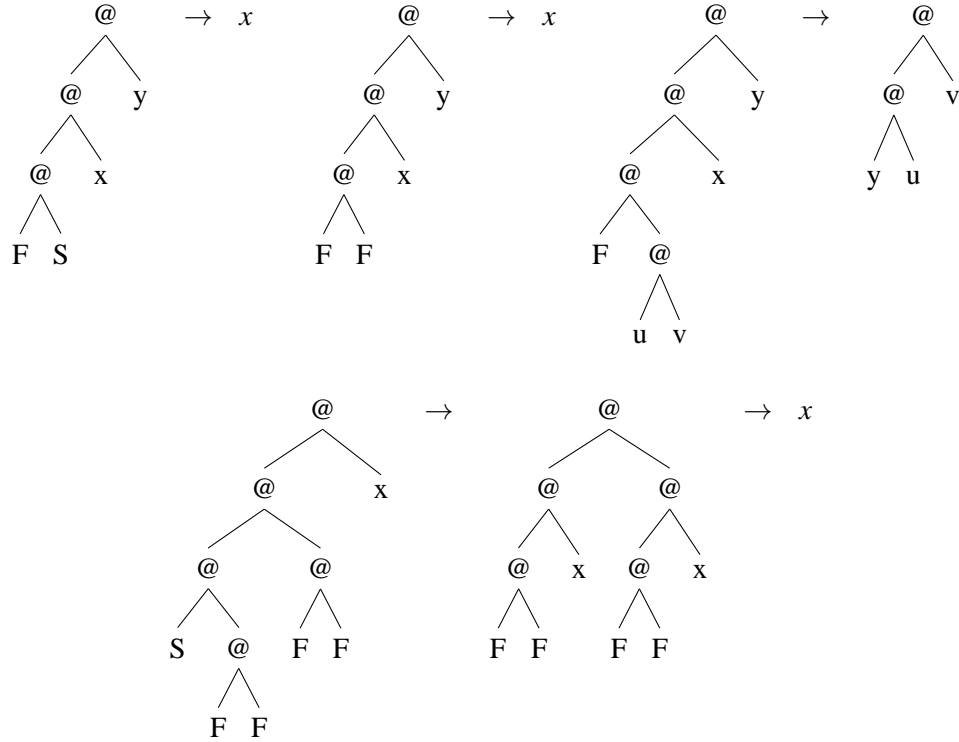


Figure 3: Terms of SF-calculus treated as trees. Above: the reduction rules for F on atoms and compound terms. Below: evaluation of the identity function $S (F F) (F F)$.

2.3 SF Combinatory Calculus

The *SF Combinatory Calculus* is a recently-developed system of combinators for expressing computation that manipulates the internal structure of programs [6]. It consists of just two combinators: S and F . S is the same as in SK-calculus. F is a *factorisation* combinator that allows non-atomic expressions to be split up into their component parts; it has two reduction rules (also depicted in Figure 3):

$$\begin{aligned} F f x y &\rightarrow x && \text{if } f = S \text{ or } f = F \\ F (u v) x y &\rightarrow y u v && \text{if } u v \text{ is a factorable form} \end{aligned}$$

A *factorable form* is a term of the form S , $S u$, $S u v$, F , $F u$ or $F u v$, for any terms u and v ; that is, a term cannot be factorised if it could be reduced at the outermost level. This ensures that reduction is globally *confluent*, regardless of the reduction order chosen. It also means that the usual notion of (weak) head reduction is not sufficient for evaluating programs in this system: if a term is of the form $F f x y$, then f must be head reduced (if possible) before applying the reduction rule for F .

F stretches our usual notion of what constitutes a combinator slightly, as it has two rewrite rules, with the conclusion of the second not being built from application of its arguments, as it deconstructs the application $u v$. Nonetheless, it is still fair to call SF-calculus a combinatory calculus, as terms in the calculus are still built solely from application of its atoms S and F .

Confluence and the theory of weak equality. Confluence means that, for any terms u , v and v' , if $u \rightarrow^* v$ and $u \rightarrow^* v'$, it follows that there is a term w with $v \rightarrow^* w$ and $v' \rightarrow^* w$. This property can be proved for SF-calculus using the standard technique of parallel reductions. The *weak equality* relation $=_w$ is the symmetric, reflexive, transitive closure of the reduction relation \rightarrow . From confluence and the fact that the terms S and F are irreducible, we can conclude that there are terms u and v such that $u \not\equiv_w v$. That is, the equational theory of $=_w$ for SF-calculus is *consistent*.

The obvious way of adding a factorisation operator to λ -calculus has no restriction to factorable forms equivalent to that for F . Consequently, adding this operator breaks confluence, so the resulting theory of weak equality is not consistent.

Extensional equality. Two terms are *extensionally equal* if they compute the same function, perhaps in different ways. Within the SF-calculus, it is possible to *distinguish* between two such terms. Consequently, SF-calculus cannot be translated into λ -calculus. For example, consider $t_1 = F F S$ and $t_2 = F S S$. For any term u , we have $t_1 u = F F S u \rightarrow^* S$ and $t_2 u = F S S u \rightarrow^* S$, so t_1 and t_2 are extensionally equal (and behave like the term $K S$ of SK-calculus). In SK-calculus or λ -calculus, if two terms t_1 and t_2 are extensionally equal, then we can replace one with the other without changing the result of a computation. However, this is not the case in SF-calculus, as we can use F to construct a term v (schematically $v = \lambda t. F t _ (\lambda u. \lambda v. F u _ (\lambda x. \lambda y. y))$) such that $v t_1 \rightarrow^* F$ and $v t_2 \rightarrow^* S$.

In SK-calculus, it is possible to extend the theory of weak equality $=_w$ with a rule corresponding to η -reduction, yielding a theory of extensional equality $=_{ext}$ such that $t_1 =_{ext} t_2$ if and only if t_1 and t_2 are extensionally equal [8]. Clearly, any reasonable attempt to extend the theory of weak equality for SF-calculus to an extensional theory of equality will be inconsistent, as it will equate S with F . This is in direct and deliberate contrast to SK-calculus.

Expressivity of SF-calculus. There is a translation from SK-calculus into SF-calculus: K can be expressed as $F F$. Hence all functions expressible in SK-calculus and thus λ -calculus are expressible in SF-calculus.

SF-calculus is *structure complete*, in the sense that it can pattern match over normal forms of terms (those having no redexes) and distinguish between any two different terms in normal form. In particular, for any two such terms t_1 and t_2 , there is a term e such that we have $e t_1 \rightarrow^* S$ and $e t_2 \rightarrow^* F$. Adding System F types to SF-calculus (and giving names to some other combinators), the resulting calculus can encode and type an interpreter for its own language [12]. Thus it presents a promising theoretical foundation for reasoning about programs that transform other programs, for example by means of partial evaluation.

As a more concrete example of the sorts of programs we might write in SF-calculus, suppose we have an expression $f x y$ and we want to flip its arguments to give $f y x$ [7]. For example, perhaps we are writing an optimising compiler, f is a commutative function that is not strict in both arguments and we expect $f y x$ to execute faster than $f x y$. Schematically, we could write a program performing this transformation as:

$$\lambda a.F a _ (\lambda b.\lambda y.F b _ (\lambda f.\lambda x.f y x))$$

where $_$ is any dummy value. Expressed purely in terms of S and F , this can be written as:

$$(SF(FFS))(FF(S(S(FF(S(FFS)(FF)))(SF(FFS)))(FF(S(FF(S(S(FF)(FF)))(FF)))))$$

Obviously, because of its lack of readability, SF-calculus (like SK-calculus and λ -calculus) is not suitable for use directly by human programmers.

3 OCFA for SK-Calculus

Before we can formulate OCFA for SF-calculus, we must first consider what it means for SK-calculus. A central idea in OCFA for λ -calculus is that the analysis computes an over-approximation of the expressions that may be bound to a variable. It seems a little perverse to apply this to SK-calculus, where there are deliberately no variables.

As SK-calculus can be translated into λ -calculus, it is easy enough to translate a term t of SK-calculus into an equivalent λ -expression $lambda(t) = e$ and analyse that. We could define our analysis by $\Gamma \models_{SK} t \iff \Gamma \models_{\lambda} lambda(t)$. Furthermore, any SK-calculus reduction $t \rightarrow t'$ corresponds to a sequence of 2 (for K) or 3 (for S) λ -calculus β -reductions. So for $lambda(t') = e'$ we have $e \rightarrow^* e'$. Then, as OCFA is coherent with evaluation following an arbitrary β -reduction strategy, we would have $\Gamma \models_{\lambda} e'$ and hence $\Gamma \models_{SK} t'$, showing the coherence of our combinatory OCFA with evaluation.

But what then is the meaning of the resulting analysis? We can answer this by reversing the translation (for example, using *unlambda*) to produce a labelled semantics for SK-calculus and OCFA rules that apply directly to SK combinatory terms.

3.1 Labelled Semantics

First we will look at the result of β -reducing expressions $lambda(S f g x)$ and $lambda(K x y)$ using the labelled semantics of λ -calculus. We begin by extending *lambda* to produce labelled expressions, as shown in Figure 4.

Here we have extended *Label* to give an easy, syntactic way of associating a fixed set of “sublabels” with each ordinary label. The choice of names for the sublabels is somewhat arbitrary, although we have chosen them to match the structure of the expressions. It is important at this stage that the label on each

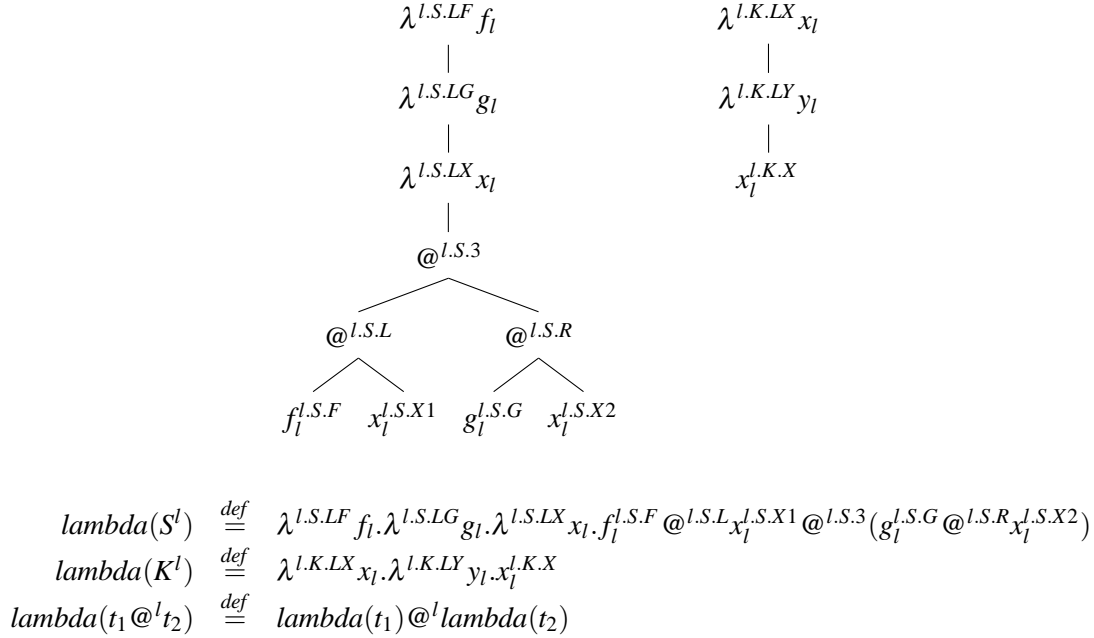


Figure 4: The labelled λ -calculus translation $\text{lambda}(t)$ (below), with $\text{lambda}(S^l)$ (upper-left) and $\text{lambda}(K^l)$ (upper-right) illustrated as trees.

expression remains distinct, so that we do not lose precision in formulating our analysis. We can now use lambda to produce labelled reduction rules for SK-calculus:

$$\begin{aligned} K^{l_2} @^{l_3} x^{l_1} @^{l_4} y^{l_0} &\rightarrow x^{l_1} \\ S^{l_3} @^{l_4} f^{l_2} @^{l_5} g^{l_1} @^{l_6} x^{l_0} &\rightarrow (f^{l_2} @^{l_3.S.L} x^{l_0}) @^{l_3.S.3} (g^{l_1} @^{l_3.S.R} x^{l_0}) \end{aligned}$$

Note that in the conclusion of the reduction of S , there are new labels that were not present in the original program. These are the sublabels from the applications introduced by lambda . In the λ -calculus formulation, these are present inside the λ -expression before reduction; the reduction exposes them. A consequence of this is that, in analysing a term of SK-calculus, we will need to consider labels that do not occur in the term. If the set of labels were infinite, this might pose a problem for an analysis. However, this is not the case, as the names of the labels are syntactically derived from the label on S ; only a finite, statically derivable set of labels may arise during the execution of a term.

3.2 Analysis Rules

We are now able to translate the rules of OCFA for λ -calculus into new rules for SK-calculus, as shown in Figure 5. Note that a label is now either a base label (as before, taken from \mathbb{N}) or a base label suffixed with a sublabel name (taken from a fixed, finite set). In performing the translation, we have eliminated some unnecessary or trivial constraints, such as those for tracking the 2nd argument to K , which is never used. We have restricted the grammar of abstract values to just instances of S and K with different numbers of arguments applied. We have also made a small change from OCFA for λ -calculus. The constraints that express the result of reducing an S are only *activated* if it is possible for that S to be reduced. This may improve precision slightly, but would be unsound in the λ -calculus setting, where we can reduce

Base Labels	$\mathbb{N} \ni n$	
Sublabel Names	$s ::= S.0 \mid S.1 \mid S.2 \mid S.3 \mid S.L \mid S.R \mid K.0$	
Labels	$Label \ni l ::= n \mid n.s$	
Labelled Terms	$t ::= S^n \mid K^n \mid t_1 @^l t_2 \mid \langle x \rangle^l$	
Abstract Values	$Abs \ni v ::= S_0^n \mid S_1^n \mid S_2^n \mid K_0^n \mid K_1^n$	
Abstract Environment	$\Gamma : Label \rightarrow \mathcal{P}(Abs)$	
Abstract Activation	$\varphi : Label \rightarrow Bool$	
$\Gamma, \varphi \models S^n$	$\iff S_0^n \in \Gamma(n) \wedge (\varphi(n) \Rightarrow \Gamma, \varphi \models t_{S^n})$	
$\Gamma, \varphi \models K^n$	$\iff K_0^n \in \Gamma(n)$	
$\Gamma, \varphi \models t_1^{l_1} @^{l_3} t_2^{l_2}$	$\iff \begin{aligned} &\Gamma, \varphi \models t_1 \wedge \Gamma, \varphi \models t_2 \\ &\wedge \forall S_0^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.S.0) \wedge S_1^n \in \Gamma(l_3) \\ &\wedge \forall S_1^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.S.1) \wedge S_2^n \in \Gamma(l_3) \\ &\wedge \forall S_2^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.S.2) \wedge \Gamma(n.S.3) \subseteq \Gamma(l_3) \wedge \varphi(n) \\ &\wedge \forall K_0^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.K.0) \wedge K_1^n \in \Gamma(l_3) \\ &\wedge \forall K_1^n \in \Gamma(l_1). \Gamma(n.K.0) \subseteq \Gamma(l_3) \end{aligned}$	
$\Gamma, \varphi \models \langle x \rangle^l$	$\iff true$	
$t_{S^n} \stackrel{def}{=} (\langle f \rangle^{n.S.0} @^{n.S.L} \langle x \rangle^{n.S.2}) @^{n.S.3} (\langle g \rangle^{n.S.1} @^{n.S.R} \langle x \rangle^{n.S.2})$		

Figure 5: OCFA for SK-calculus

the expression corresponding to S even if it only has 1 or 2 arguments. It will be more important for SF-calculus. In order to track whether constraints for an instance have been activated, we introduce a new component $\varphi : Label \rightarrow Bool$ to the solution of the constraints, with $\varphi(n)$ being true when the constraints for S^n are active.

The intuitive meaning of $S_0^n \in \Gamma(l)$ is that S^n may occur at the point labelled l , hence the rule for $\Gamma, \varphi \models S^n$. The meaning of $S_1^n \in \Gamma(l)$ is that a term built from applying 1 argument to S^n may occur at l , or that S^n may occur as the 1st left child of the term tree node labelled l . The meaning of S_2^n is analogous (but for 2 arguments or the 2nd left child), as is that of K_0^n and K_1^n (but for K , not S).

The abstract values in $\Gamma(n.S.0)$ are meant to over-approximate the values that may occur as the 1st argument to S^n ; similarly for $\Gamma(n.S.1)$ and the 2nd argument, and analogously for $n.S.2$ and $n.K.0$.

This leads to the explanation of the conjunction of conditions for $\Gamma, \varphi \models t_1^{l_1} @^{l_3} t_2^{l_2}$. For example, the condition involving $\forall S_0^n$ ensures that, if S^n may occur in function position, then: the abstraction $\Gamma(n.S.0)$ of the 1st argument of S^n over-approximates the arguments that may be supplied by t_2 ; and the result of the application needs only 2 more arguments for a reduction to occur. The condition involving $\forall S_1^n$ and the first part of the condition with $\forall S_2^n$ are similar. The condition on $\forall K_0^n$ is analogous to that for $\forall S_0^n$. The condition with $\forall K_1^n$ simply says that the result of reducing K may be anything that occurs as its 1st argument.

The second part of the condition on $\forall S_2^n$ is more complicated. In the event that S^n may receive 3 arguments and hence be reduced, it introduces constraints for the conclusion of the reduction, which are those generated by analysis of the constant applicative term t_{S^n} . It also says that the result of the reduction may be anything that occurs at the root of that term, which has label $n.S.3$.

The introduction of the constraints for t_{S^n} is forced by asserting $\varphi(n)$, which produces the corresponding constraints in the rule for $\Gamma, \varphi \models S^n$. The use of φ avoids the introduction of a recursive loop

$\Gamma(0) = \{K_0^0\}$	$\Gamma(1) = \{K_0^1\}$	$\Gamma(2) = \{S_0^2\}$	$\Gamma(2.S.0) = \{K_0^1\}$
$\Gamma(2.S.1) = \{K_0^0\}$	$\Gamma(3) = \{S_1^2\}$	$\Gamma(4) = \{S_2^2\}$	$\Gamma(5) = \{K_0^5\}$
$\Gamma(5.K.0) = \{S_2^2\}$	$\Gamma(6) = \{K_0^6\}$	$\Gamma(6.K.0) = \{S_2^2\}$	$\Gamma(7) = \{S_0^7\}$
$\Gamma(7.S.0) = \{K_0^6\}$	$\Gamma(7.S.1) = \{K_0^5\}$	$\Gamma(7.S.2) = \{S_2^2\}$	$\Gamma(7.S.3) = \{S_2^2\}$
$\Gamma(7.S.L) = \{K_1^6\}$	$\Gamma(7.S.R) = \{K_1^5\}$	$\Gamma(8) = \{S_1^7\}$	$\Gamma(9) = \{S_2^7\}$
$\Gamma(10) = \{S_2^2\}$	$\varphi(7) = \text{true}$		

$$\Gamma, \varphi \models (S^7 @^8 K^6 @^9 K^5) @^{10} (S^2 @^3 K^1 @^4 K^0)$$

Figure 6: Solution of the analysis for application of identity to itself in SK-calculus.

in the constraint rules; an alternative method would be to use a coinductive definition of \models . Within t_{S^n} we use dummy terms of the form $\langle x \rangle^l$ to give the leaf node of the term tree a label; here f , g and x have no meaning (other than to make reading the rules easier) and play no role in the analysis.

This analysis may seem like a step backwards, as we have replaced a small set of general rules for λ -calculus with a larger, more specific set of rules for SK-calculus. However, there are a number of benefits. Firstly, the rules for S can be used directly in OCFA for SF-calculus. Secondly, they reveal the meaning of OCFA in SK-calculus: the abstract values at a labelled point tell us which combinators may occur at that point *and* locally at its left branches. This insight will be key in both producing an accurate analysis for F and for justifying why it is reasonable to call that analysis OCFA. Finally, because SK-calculus does not have to deal with arbitrary substitution or the intricacies of name-binding, the proof of correctness for this system is considerably simpler than that for λ -calculus.

Recall the example of OCFA for λ -calculus involving applying the identity function to itself. The corresponding SK-calculus term and a solution for its analysis are shown in Figure 6. Note that $\Gamma(10) = \Gamma(4) = \{S_2^2\}$, indicating that the result of evaluation has S^2 as its second left child; that is, it is the second identity function $(S^2 @^3 K^1 @^4 K^0)$.

3.3 Correctness

We now prove the correctness of this analysis for SK-calculus. First we make some observations about the satisfaction of constraints:

Lemma 1 (SK Substitution). *If $\Gamma, \varphi \models t_1^{l_1} @^{l_2} t_2^{l_2}$, as well as $\Gamma, \varphi \models t_1'^{l_1}$ and $\Gamma, \varphi \models t_2'^{l_2}$, with $\Gamma(l_1') \subseteq \Gamma(l_1)$, $\Gamma(l_2') \subseteq \Gamma(l_2)$ and $\Gamma(l_3') \supseteq \Gamma(l_3)$ then $\Gamma, \varphi \models t_1'^{l_1} @^{l_2'} t_2'^{l_2}$.*

Proof. Trivial by inspection of the constraints generated by @. □

Lemma 2 (SK Reduction Coherence). *For any top-level reduction $t^l \rightarrow t'^{l'}$, if $\Gamma, \varphi \models t^l$ then 1) $\Gamma, \varphi \models t'^{l'}$ and 2) $\Gamma(l') \subseteq \Gamma(l)$.*

Proof. Case split on the two kinds of top-level reduction (S and K).

Case S: We have $t^l = S^{l_3} @^{l_4} f^{l_2} @^{l_5} g^{l_1} @^{l_6} x^{l_0}$ and $t'^{l'} = (f^{l_2} @^{l_3.S.L} x^{l_0}) @^{l_3.S.3} (g^{l_1} @^{l_3.S.R} x^{l_0})$. Expanding the constraints for $\Gamma, \varphi \models t^l$, we have: $S_0^{l_3} \in \Gamma(l_3)$; $S_1^{l_3} \in \Gamma(l_4)$; $S_2^{l_3} \in \Gamma(l_5)$; hence $\Gamma(l_3.S.3) \subseteq \Gamma(l_6)$ (proving Condition 2) and $\varphi(l_3)$ is true. As $\Gamma, \varphi \models S_0^{l_3}$ and $\varphi(l_3)$, we have $\Gamma, \varphi \models t_{S_{l_3}}$. But $t_{S_{l_3}}$ can be turned into t' by substituting f , g and x at its leaves. So to prove Condition 1, we just need to show that we can use the Substitution Lemma. Now as $\Gamma, \varphi \models t^l$, we get: $\Gamma, \varphi \models f^{l_2}$; $\Gamma, \varphi \models g^{l_1}$; and $\Gamma, \varphi \models x^{l_0}$.

Furthermore: from $S_0^{l_3} \in \Gamma(l_3)$ we get $\Gamma(l_2) \subseteq \Gamma(l_3.S.0)$; from $S_1^{l_3} \in \Gamma(l_4)$ we get $\Gamma(l_1) \subseteq \Gamma(l_3.S.1)$; and from $S_2^{l_3} \in \Gamma(l_3)$ we get $\Gamma(l_0) \subseteq \Gamma(l_3.S.2)$. So the Substitution Lemma can be used to prove Condition 1.

Case K: We have $t^l = K^{l_2} @^{l_3} x^{l_1} @^{l_4} y^{l_0}$ and $t^{l'} = x^{l_1}$. From $\Gamma, \varphi \models t^l$ we have $\Gamma, \varphi \models x^{l_1}$, showing Condition 1. Expanding the constraints for $\Gamma, \varphi \models t^l$ further, we get: $K_0^{l_2} \in \Gamma(l_2)$; hence $\Gamma(l_1) \subseteq \Gamma(l_2.K.0)$ and $K_1^{l_2} \in \Gamma(l_3)$; thus $\Gamma(l_2.K.0) \subseteq \Gamma(l_4)$. Combining these gives $\Gamma(l_1) \subseteq \Gamma(l_4)$, proving Condition 2. \square

Theorem 1 (SK Evaluation Coherence). *For any reduction in context $C[t^{l_1}]^{l_2} \rightarrow C[t^{l'_1}]^{l'_2}$, if $\Gamma, \varphi \models C[t^{l_1}]^{l_2}$ then 1) $\Gamma, \varphi \models C[t^{l'_1}]^{l'_2}$ and 2) $\Gamma(l'_2) \subseteq \Gamma(l_2)$.*

Proof. For the empty context, this follows immediately from the Reduction Coherence Lemma. For a non-empty context C , Condition 2 is trivially true as $l'_2 = l_2$. For Condition 1, the reduction occurs at either the left child or right child of an application node in the term tree (as all other nodes are leaves). Any constraints generated by the context are unchanged and hence remain satisfied. For the hole in the context, from $\Gamma, \varphi \models C[t^{l_1}]^{l_2}$ we have $\Gamma, \varphi \models t^{l_1}$, so by Reduction Coherence we get $\Gamma, \varphi \models t^{l'_1}$ with $\Gamma(l'_1) \subseteq \Gamma(l_1)$. Hence any constraints within $t^{l'_1}$ are satisfied. That just leaves the constraints generated by the interaction between the application at the hole of the context and t^l . We can apply the Substitution Lemma to the application node to show that they are satisfied, which gives Condition 1 as required. \square

Corollary 1 (SK Soundness). *If $\Gamma, \varphi \models t^l$ and $t \rightarrow^* t'$ then $\Gamma, \varphi \models t'^l$.*

Proof. By induction over the length of the derivation of \rightarrow^* and application of the Evaluation Coherence Theorem. \square

4 OCFA for SF-Calculus

We now turn our attention to formulation of OCFA for SF-calculus. As F is not encodable in λ -calculus, we cannot argue for the correctness of our analysis by appeal to the translation *lambda*. Instead, we must follow the style of our formulation for SK-calculus.

4.1 Labelled Semantics

Following the labelled reduction for S , we introduce the following labelled reductions for F :

$$\begin{aligned} F^{l_3} @^{l_4} f^{l_2} @^{l_5} x^{l_1} @^{l_6} y^{l_0} &\rightarrow x^{l_1} && \text{if } f = S \text{ or } f = K \\ F^{l_3} @^{l_4} (u^{l_7} @^{l_2} v^{l_8}) @^{l_5} x^{l_1} @^{l_6} y^{l_0} &\rightarrow (y^{l_0} @^{l_3.F.M} u^{l_7}) @^{l_3.F.3} v^{l_8} && \text{if } u \text{ } v \text{ is a factorable form} \end{aligned}$$

4.2 Analysis Rules

There are two main problems to consider in analysing F : how to determine whether the 1st argument is a factorable form and, when that argument is a factorable form, how to deconstruct its abstract representation.

Concerning the first problem, if we think back to our analysis for SK-calculus, a term t^l might evaluate to an atom S^n or F^n if (for some n) $S_0^n \in \Gamma(l)$ or $K_0^n \in \Gamma(l)$. Its normal form might be a non-atomic term if $\Gamma(l)$ contains any other abstract values. We can use the same idea for SF-calculus, except with F_0^n in place of K_0^n .

Base Labels	$\mathbb{N} \ni n$
Sublabel Names	$s ::= S.0 \mid S.1 \mid S.2 \mid S.3 \mid S.L \mid S.R \mid F.0 \mid F.1 \mid F.2 \mid F.3 \mid F.L \mid F.R \mid F.M$
Labels	$Label \ni l ::= n \mid n.s$
Labelled Terms	$t ::= S^n \mid F^n \mid t_1 @^l t_2 \mid \langle x \rangle^l$
Abstract Values	$Abs \ni v ::= S_0^n \mid S_1^n \mid S_2^n \mid F_0^n \mid F_1^n \mid F_2^n \mid @^{(l_1, l_2)}$
Abstract Environment	$\Gamma : Label \rightarrow \mathcal{P}(Abs)$
Abstract Activation	$\varphi : Label \rightarrow Bool$
$\Gamma, \varphi \models S^n$	$\iff S_0^n \in \Gamma(n) \wedge (\varphi(n) \Rightarrow \Gamma, \varphi \models t_{S^n})$
$\Gamma, \varphi \models F^n$	$\iff F_0^n \in \Gamma(n) \wedge \varphi(n) \Rightarrow (\exists n_0. S_0^{n_0} \in \Gamma(n.F.0) \vee F_0^{n_0} \in \Gamma(n.F.0)) \Rightarrow \Gamma(n.F.1) \subseteq \Gamma(n.F.3) \wedge \varphi(n) \Rightarrow (\exists n_0. S_1^{n_0} \in \Gamma(n.F.0) \vee S_2^{n_0} \in \Gamma(n.F.0) \vee F_1^{n_0} \in \Gamma(n.F.0) \vee F_2^{n_0} \in \Gamma(n.F.0)) \Rightarrow \Gamma, \varphi \models t_{F^n} \wedge \forall @^{l_1, l_2} \in \Gamma(n.F.0). \Gamma(l_1) \subseteq \Gamma(n.F.L) \wedge \Gamma(l_2) \subseteq \Gamma(n.F.R)$
$\Gamma, \varphi \models t_1^{l_1} @^{l_3} t_2^{l_2}$	$\iff \Gamma, \varphi \models t_1 \wedge \Gamma, \varphi \models t_2 \wedge \exists @^{l_4, l_5} \in \Gamma(l_3). \Gamma(l_1) \subseteq \Gamma(l_4) \wedge \Gamma(l_2) \subseteq \Gamma(l_5) \wedge \forall S_0^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.S.0) \wedge S_1^n \in \Gamma(l_3) \wedge \forall S_1^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.S.1) \wedge S_2^n \in \Gamma(l_3) \wedge \forall S_2^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.S.2) \wedge \Gamma(n.S.3) \subseteq \Gamma(l_3) \wedge \varphi(n) \wedge \forall F_0^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.F.0) \wedge F_1^n \in \Gamma(l_3) \wedge \forall F_1^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.F.1) \wedge F_2^n \in \Gamma(l_3) \wedge \forall F_2^n \in \Gamma(l_1). \Gamma(l_2) \subseteq \Gamma(n.F.2) \wedge \Gamma(n.F.3) \subseteq \Gamma(l_3) \wedge \varphi(n)$
$\Gamma, \varphi \models \langle x \rangle^l$	$\iff true$
t_{S^n}	$\stackrel{def}{=} (\langle f \rangle^{n.S.0} @^{n.S.L} \langle x \rangle^{n.S.2}) @^{n.S.3} (\langle g \rangle^{n.S.1} @^{n.S.R} \langle x \rangle^{n.S.2})$
t_{F^n}	$\stackrel{def}{=} (\langle y \rangle^{n.F.2} @^{n.F.M} \langle u \rangle^{n.F.L}) @^{n.F.3} \langle v \rangle^{n.F.R}$

Figure 7: OCFA for SF-calculus

As for the second problem, in order to deconstruct abstract values, we introduce a new type of abstract value $@^{l_1, l_2}$. Intuitively, the abstract value indicates that any concrete value was produced by applying a term approximated by $\Gamma(l_1)$ to a term approximated by $\Gamma(l_2)$. The resulting analysis is shown in Figure 7.

We have reused the analysis rules for S . The rules for F are mostly very similar. This is to be expected, as they both take 3 arguments. In the rule for $\Gamma, \varphi \models F^n$, there are two separate sets of constraints that can be activated by $\varphi(n)$, corresponding to the two reduction rules. Both involve a further condition that corresponds to testing whether the 1st argument may be atomic or a factorable form. The conclusion to the first, corresponding to the atomic case, is similar to the last rule for K in $\Gamma, \varphi \models t_1^{l_1} @^{l_3} t_2^{l_2}$ of the analysis for SK-calculus. The conclusion to the second, which handles factorisation, introduces the constraints generated by the applicative term t_{F^n} in a similar style to the case for S and t_{S^n} . However, it also adds new constraints to t_{F^n} corresponding to the factorisation of the 1st argument.

There is also a new constraint for $\Gamma, \varphi \models t_1^{l_1} @^{l_3} t_2^{l_2}$ that introduces abstract values of the form $@^{l_1, l_2}$. When analysing a term, this is easily satisfied by setting $@^{l_1, l_2} \in \Gamma(l_3)$. The slightly more complicated constraint here is necessary to ensure coherence of the analysis with evaluation.

A term t^l can be analysed by finding a Γ and φ such that $\Gamma, \varphi \models t^l$. This is done by solving the

$$\begin{array}{ll}
\Gamma(0) &= \{F_0^0\} \\
\Gamma(1.F.0) &= \{F_0^0\} \\
\Gamma(3) &= \{F_0^3\} \\
\Gamma(4.F.0) &= \{F_0^3\} \\
\Gamma(6) &= \{S_0^6\} \\
\Gamma(6.S.1) &= \{F_1^1, @^{(1,0)}\} \\
\Gamma(8) &= \{S_2^6, @^{(7,2)}\} \\
\Gamma(10) &= \{F_0^{10}\} \\
\Gamma(10.F.1) &= \{S_2^6, @^{(7,2)}\} \\
\Gamma(12) &= \{F_0^{12}\} \\
\Gamma(13.F.0) &= \{F_0^{12}\} \\
\Gamma(13.F.3) &= \{S_2^6, @^{(7,2)}\} \\
\Gamma(14) &= \{F_1^{13}, @^{(13,12)}\} \\
\Gamma(15.S.0) &= \{F_1^{13}, @^{(13,12)}\} \\
\Gamma(15.S.2) &= \{S_2^6, @^{(7,2)}\} \\
\Gamma(15.S.L) &= \{F_2^{13}, @^{(15.S.0,15.S.2)}\} \\
\Gamma(15.S.R) &= \{F_2^{10}, @^{(15.S.1,15.S.2)}\} \\
\Gamma(18) &= \{S_2^6, @^{(15.S.L,15.S.R)}, @^{(17,8)}, @^{(7,2)}\} \\
\varphi(13) &= \text{true} \\
\Gamma(1) &= \{F_0^1\} \\
\Gamma(2) &= \{F_1^1, @^{(1,0)}\} \\
\Gamma(4) &= \{F_0^4\} \\
\Gamma(5) &= \{F_1^4, @^{(4,3)}\} \\
\Gamma(6.S.0) &= \{F_1^4, @^{(4,3)}\} \\
\Gamma(7) &= \{S_1^6, @^{(6,5)}\} \\
\Gamma(9) &= \{F_0^9\} \\
\Gamma(10.F.0) &= \{F_0^9\} \\
\Gamma(11) &= \{F_1^{10}, @^{(10,9)}\} \\
\Gamma(13) &= \{F_0^{13}\} \\
\Gamma(13.F.1) &= \{S_2^6, @^{(7,2)}\} \\
\Gamma(13.F.2) &= \{F_2^{10}, @^{(15.S.1,15.S.2)}\} \\
\Gamma(15) &= \{S_0^{15}\} \\
\Gamma(15.S.1) &= \{F_1^{10}, @^{(10,9)}\} \\
\Gamma(15.S.3) &= \{S_2^6, @^{(15.S.L,15.S.R)}, @^{(7,2)}\} \\
\Gamma(16) &= \{S_1^{15}, @^{(15,14)}\} \\
\Gamma(17) &= \{S_2^{15}, @^{(16,11)}\} \\
\varphi(15) &= \text{true}
\end{array}$$

$$\Gamma, \varphi \models (S^{15} @^{16} (F^{13} @^{14} F^{12}) @^{17} (F^{10} @^{11} F^9)) @^{18} (S^6 @^7 (F^4 @^5 F^3) @^8 (F^1 @^2 F^0))$$

Figure 8: Solution of the analysis for application of identity to itself in SF-calculus.

constraints using a fixed point process, much as with OCFA for λ -calculus. We need only consider $\mathcal{O}(n)$ abstract values (corresponding to nodes in the term tree of t), so we retain the polynomial time complexity of OCFA.

Consider once again the example of applying the identity function to itself. The corresponding SF-calculus term and its analysis are shown in Figure 8; note that $S_2^6 \in \Gamma(18)$, indicating the result correctly.

4.3 Correctness

Correctness of the analysis follows by the same sequence of results as for SK-calculus.

Lemma 3 (SF Substitution). *If $\Gamma, \varphi \models t_1^{l_1} @^{l_3} t_2^{l_2}$, as well as $\Gamma, \varphi \models t_1'^{l'_1}$ and $\Gamma, \varphi \models t_2'^{l'_2}$, with $\Gamma(l'_1) \subseteq \Gamma(l_1)$, $\Gamma(l'_2) \subseteq \Gamma(l_2)$ and $\Gamma(l'_3) \supseteq \Gamma(l_3)$ then $\Gamma, \varphi \models t_1'^{l'_1} @^{l'_3} t_2'^{l'_2}$.*

Proof. Again, trivial by inspection of the constraints generated by $@$. It is at this point that the correct formulation of the constraint $\exists @^{l_4, l_5} \in \Gamma(l_3). \Gamma(l_1) \subseteq \Gamma(l_4) \wedge \Gamma(l_2) \subseteq \Gamma(l_5)$ is important; the lemma does not hold if we use the simpler constraint $@^{l_1, l_2} \in \Gamma(l_3)$. \square

Lemma 4 (SF Reduction Coherence). *For any top-level reduction $t^l \rightarrow t'^{l'}$, if $\Gamma, \varphi \models t^l$ then 1) $\Gamma, \varphi \models t'^{l'}$ and 2) $\Gamma(l') \subseteq \Gamma(l)$.*

Proof. Case split on the two kinds of top-level reduction (S and F).

Case S: Largely as for SK-calculus. The only new point is that we must check the constraints on the abstract $@$ values generated by $@^{l_3.S.L}$ and $@^{l_3.S.R}$ still hold.

Case F: We have $t^l = F^{l_3} @^{l_4} f^{l_2} @^{l_5} x^{l_1} @^{l_6} y^{l_0}$. We begin as for S by expanding the constraints for $\Gamma, \varphi \models t^l$. We get $F_0^{l_3} \in \Gamma(l_3)$, $F_1^{l_3} \in \Gamma(l_4)$ and $F_2^{l_3} \in \Gamma(l_5)$; also $\Gamma(l_2) \subseteq \Gamma(l_3.F.0)$, $\Gamma(l_1) \subseteq \Gamma(l_3.F.1)$ and $\Gamma(l_0) \subseteq \Gamma(l_3.F.2)$; as well as $\Gamma(l_3.F.3) \subseteq \Gamma(l_6)$ and $\varphi(l_3)$. Now there are two subcases depending on whether f is factorable.

Subcase f is not factorable: We have $t^{l'} = x^{l_1}$. From $\Gamma, \varphi \models t^l$ we get $\Gamma, \varphi \models x^{l_1}$, proving Condition 1. As f^{l_2} is not factorable, either $f = F^{l_2}$ and $F_0^{l_2} \in \Gamma(l_2) \subseteq \Gamma(l_3.F.0)$ or $f = S^{l_2}$ and $S_0^{l_2} \in \Gamma(l_2) \subseteq \Gamma(l_3.F.0)$. In either case, noting we already have $\Gamma, \varphi \models F^{l_3}$ and $\varphi(l_3)$, we get $\Gamma(l_3.F.1) \subseteq \Gamma(l_3.F.3)$. Combining this with $\Gamma(l_1) \subseteq \Gamma(l_3.F.1)$ and $\Gamma(l_3.F.3) \subseteq \Gamma(l_6)$ gives $\Gamma(l_1) \subseteq \Gamma(l_6)$, proving Condition 2.

Subcase f is factorable: We have $f^{l_2} = u^{l_7} @^{l_2} v^{l_8}$ and $t^{l'} = (y^{l_0} @^{l_3.F.M} u^{l_7}) @^{l_3.F.3} v^{l_8}$. Condition 2 now follows immediately from $\Gamma(l_3.F.3) \subseteq \Gamma(l_6)$. As f is factorable, either u^{l_7} or its left child w^{l_9} (if it has one) is S or F . Hence one of $F_1^{l_7}$, $S_1^{l_7}$, $F_2^{l_9}$ and $F_2^{l_9}$ must be in $\Gamma(l_3.F.0)$. Noting $\Gamma, \varphi \models F^{l_3}$ and $\varphi(l_3)$, we now have $\Gamma, \varphi \models t_{F^n}$, as well as a constraint relating abstract $@$ values in $\Gamma(l_3.F.0)$ with $\Gamma(l_3.F.L)$ and $\Gamma(l_3.F.R)$. Similarly to the case for S , in order to prove Condition 1, we note that we can obtain $t^{l''}$ by substituting y^{l_0} , u^{l_7} and v^{l_8} into t_{F^n} , so we need to show that the Substitution Lemma is applicable. For y^{l_0} this is easy, as we already have $\Gamma(l_0) \subseteq \Gamma(l_3.F.2)$. For u^{l_7} and v^{l_8} , there must exist some $@^{l_A, l_B} \in \Gamma(l_2)$ with $\Gamma(l_7) \subseteq \Gamma(l_A)$ and $\Gamma(l_8) \subseteq \Gamma(l_B)$. But $\Gamma(l_2) \subseteq \Gamma(l_3.F.0)$, so $@^{l_A, l_B} \in \Gamma(l_3.F.0)$. Then, using the above constraint on abstract $@$ values, $\Gamma(l_A) \subseteq \Gamma(l_3.F.L)$ and $\Gamma(l_B) \subseteq \Gamma(l_3.F.R)$. Hence $\Gamma(l_7) \subseteq \Gamma(l_3.F.L)$ and $\Gamma(l_8) \subseteq \Gamma(l_3.F.R)$, so we can apply the Substitution Lemma to prove Condition 1. \square

Theorem 2 (SF Evaluation Coherence). *For any reduction in context $C[t^l]^{l_2} \rightarrow C[t^{l'}]^{l'_2}$, if $\Gamma, \varphi \models C[t^l]^{l_2}$ then 1) $\Gamma, \varphi \models C[t^{l'}]^{l'_2}$ and 2) $\Gamma(l'_2) \subseteq \Gamma(l_2)$.*

Proof. The proof is as for SK-calculus. The only point of note is that the constraints between the application at the hole of the context and t^l now include a constraint on an abstract $@$ value. However, this is still handled by using the Substitution Lemma. \square

Corollary 2 (SF Soundness). *If $\Gamma, \varphi \models t^l$ and $t \rightarrow^* t'$ then $\Gamma, \varphi \models t'^l$.*

Proof. As for SK-calculus. \square

5 Evaluation

It is currently difficult to evaluate meaningfully the usefulness of this analysis. If one wishes to evaluate an analysis for untyped λ -calculus, then by using the usual Church encodings for numbers, lists and other datatypes, one can easily test it against examples from any textbook on functional programming. Similarly, using the translation *unlambda*, it is not much harder to evaluate an analysis for SK-calculus in this way.

There is a straightforward translation from SK-calculus to SF-calculus: simply replace K with FF . It is easy for our analysis to determine that the only possible first argument to the first F is just F , and hence that it will never be factorable. This activates constraints that are very similar to those for K in the analysis for SK-calculus. Thus it makes no difference to the precision of OCFA whether it is done on a term of SK-calculus or the same term translated into SF-calculus.

While this is encouraging in that it suggests it is reasonable to refer to our analysis as OCFA, it does not really tell us anything interesting about the precision of the analysis. The translated program does not use the power of factorisation in a meaningful way, or indeed (considering that only one reduction of F is used) at all. There is no interesting suite of programs written in SF-calculus against which to test the analysis; nor is there any existing idiomatic translation from any higher level language to SF-calculus.

If we consider only programs that do not deconstruct code (such as straightforward translations of SK-calculus programs), our analysis has the same strengths and weaknesses as other forms of OCFA: it can analyse some higher order control flow within a program, but loses precision when the same function is used in two different contexts.

If we consider programs that do inspect and manipulate the internal structure of code, there are three further places where we can lose precision. Firstly, we cannot always tell whether an argument to F will be factorable or not and in this case, we over-approximate its behaviour to cover both cases. Our technique essentially works by tracking how many arguments a combinator has been given. This is unlikely to work well when a term is simultaneously used recursively and partially applied. Secondly, when we abstractly factorise a term, we lack any contextual information, so if two applications flow into the same factorisation, we will conflate their factors. This is similar to the imprecision introduced by lack of context when using the same function in two different places in ordinary OCFA. Finally, while we make a reasonable attempt to track reduction of a term for the purpose of determining whether its normal form is an atom, we have no way of discarding non-normal forms when we factorise abstractly, so we may consider the factorisation of terms that are not factorable forms.

6 Related Work

OCFA and other forms of control flow analysis have been widely studied; see the work of Midtgaard [17] for a detailed survey.

To our knowledge, this is the first static analysis for SF-calculus. There has been some work on analysing other styles of metaprogramming. For example, Choi and others [4] consider how to analyse a form of extensional metaprogramming called staged metaprogramming, which captures the composition of code templates. They suggest using an unstaging translation that turns the metaprogramming constructs into function abstraction and record lookup, then using other existing analyses. Our own work considers how to formulate OCFA in a dynamically typed language with staged metaprogramming and variable capture [14], with a view to analysing JavaScript's eval construct [15]. In a statically typed setting, Berger and Tratt develop a Hoare-style logic [1] for a language with staged metaprogramming.

Intensional metaprogramming has often been ignored because of its semantic difficulties, or because of the dominance of the idea that extensionally equal programs ought to be indistinguishable [11]. ReFLect [7], a functional programming language for hardware design and theorem proving, allows deconstruction of code values, but this causes difficulties for its type system, even in a combinatory fragment of the language [16].

The idea that program code can be deconstructed and that its structure can influence the control flow of a program is conceptually similar to the functional programming idiom of defining functions by pattern-matching over algebraic datatypes. There has been some work on analysing functional programs from this perspective. For example, Jones and Andersen present an analysis that uses tree grammars to over-approximate the structure of data values that may be produced by a program [13]. Ong and Ramsay suggest a formalism called Pattern Matching Recursion Schemes that captures the idea in a typed setting and develop a powerful analysis for it [19].

7 Future Work and Conclusions

We have presented the first static analysis for SF-calculus, a formalism which presents a promising foundation for writing programs that transform other programs. We have proved correctness of the

analysis and shown that is comparable to standard OCFA for programs that do not rely on the ability of F to factor terms, such as those translated directly from SK-calculus. From here, there are a number of obvious directions in which to proceed.

Firstly, in order to evaluate the usefulness of the analysis and to advance our understanding of program transformation, it would be good to develop a translation from a higher level language that supports intensional metaprogramming into SF-calculus. The translation should map code deconstruction to factorisation using F .

Secondly, there is scope to improve the precision of the analysis. For standard OCFA, tracking context in the style of k -CFA or a pushdown analysis in the style of CFA2 can improve precision significantly. The same techniques may be applicable here. It may also be possible to use techniques from analysing pattern matching and tree datatypes in functional programming languages to analyse the term trees that constitute programs in SF-calculus and their pattern-matching and deconstruction with F . However, an important consideration in applying any such technique to SF-calculus would be the need to distinguish between a non-factorable term t and the factorable term t' to which it may reduce.

Finally, OCFA is often useful not as an end to itself, but because it can be combined with other analysis techniques, for example drawn from abstract interpretation, in order to improve their precision by reducing the number of execution paths or reduction sequences that must be considered to over-approximate the behaviour of a program. It would be interesting to see if, combined with such techniques, this analysis can actually be used to verify properties of programs that perform program transformations.

References

- [1] Martin Berger & Laurence Tratt (2010): *Program Logics for Homogeneous Meta-programming*. In Edmund M. Clarke & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers, Lecture Notes in Computer Science 6355*, Springer, pp. 64–81, doi:10.1007/978-3-642-17511-4_5.
- [2] Lars Bergstrom, Matthew Fluet, Matthew Le, John H. Reppy & Nora Sandler (2014): *Practical and effective higher-order optimizations*. In Johan Jeuring & Manuel M. T. Chakravarty, editors: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, ACM, pp. 81–93, doi:10.1145/2628136.2628153.
- [3] Swarat Chaudhuri (2008): *Subcubic algorithms for recursive state machines*. In George C. Necula & Philip Wadler, editors: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, pp. 159–169, doi:10.1145/1328438.1328460.
- [4] Wontae Choi, Baris Aktemur, Kwangkeun Yi & Makoto Tatsuta (2011): *Static analysis of multi-staged programs via unstaging translation*. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pp. 81–92, doi:10.1145/1926385.1926397.
- [5] T. J.W. Clarke, P. J.S. Gladstone, C. D. MacLean & A. C. Norman (1980): *SKIM - The S, K, I Reduction Machine*. In: *Proceedings of the 1980 ACM Conference on LISP and Functional Programming, LFP '80*, ACM, New York, NY, USA, pp. 128–135, doi:10.1145/800087.802798.
- [6] Thomas Given-Wilson & Barry Jay (2011): *A combinatory account of internal structure*. *J. Symb. Log.* 76(3), pp. 807–826, doi:10.2178/jsl/1309952521.
- [7] Jim Grundy, Thomas F. Melham & John W. O'Leary (2006): *A reflective functional language for hardware design and theorem proving*. *J. Funct. Program.* 16(2), pp. 157–196, doi:10.1017/S0956796805005757.
- [8] J. Roger Hindley & Jonathan P. Seldin (2008): *Lambda-Calculus and Combinators: An Introduction*, 2nd edition. Cambridge University Press, New York, NY, USA, doi:10.1017/CB09780511809835.

- [9] David Van Horn & Harry G. Mairson (2008): *Deciding kCFA is complete for EXPTIME*. In James Hook & Peter Thiemann, editors: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, ACM, pp. 275–282, doi:10.1145/1411204.1411243.
- [10] David Van Horn & Harry G. Mairson (2008): *Flow Analysis, Linearity, and PTIME*. In María Alpuente & Germán Vidal, editors: *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings, Lecture Notes in Computer Science 5079*, Springer, pp. 255–269, doi:10.1007/978-3-540-69166-2_17.
- [11] Barry Jay & Jose Vergara (2014): *Confusion in the Church-Turing Thesis*. CoRR abs/1410.7103. Available at <http://arxiv.org/abs/1410.7103>.
- [12] C. Barry Jay & Jens Palsberg (2011): *Typed self-interpretation by pattern matching*. In Manuel M. T. Chakravarty, Zhenjiang Hu & Olivier Danvy, editors: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, ACM, pp. 247–258, doi:10.1145/2034773.2034808.
- [13] Neil D. Jones & Nils Andersen (2007): *Flow analysis of lazy higher-order functional programs*. *Theor. Comput. Sci.* 375(1-3), pp. 120–136, doi:10.1016/j.tcs.2006.12.030.
- [14] Martin Lester, Luke Ong & Max Schäfer (2013): *Information Flow Analysis for a Dynamically Typed Language with Staged Metaprogramming*. In: *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, IEEE, pp. 209–223, doi:10.1109/CSF.2013.21.
- [15] Martin Mariusz Lester (2013): *Position paper: the science of boxing*. In Prasad Naldurg & Nikhil Swamy, editors: *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*, ACM, pp. 83–88, doi:10.1145/2465106.2465120.
- [16] Tom Melham, Raphael Cohn & Ian Childs (2013): *On the Semantics of ReFLect as a Basis for a Reflective Theorem Prover*. CoRR abs/1309.5742. Available at <http://arxiv.org/abs/1309.5742>.
- [17] Jan Midtgaard (2012): *Control-flow analysis of functional programs*. *ACM Comput. Surv.* 44(3), p. 10, doi:10.1145/2187671.2187672.
- [18] Flemming Nielson, Hanne Riis Nielson & Chris Hankin (1999): *Principles of program analysis*. Springer, doi:10.1007/978-3-662-03811-6.
- [19] C.-H. Luke Ong & Steven J. Ramsay (2011): *Verifying higher-order functional programs with pattern-matching algebraic data types*. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pp. 587–598, doi:10.1145/1926385.1926453.
- [20] Olin Shivers (1988): *Control-Flow Analysis in Scheme*. In Richard L. Wexelblat, editor: *Proceedings of the ACM SIGPLAN’88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, ACM, pp. 164–174, doi:10.1145/53990.54007.
- [21] David A Turner (1979): *Another algorithm for bracket abstraction*. *The Journal of Symbolic Logic* 44(02), pp. 267–270, doi:10.2307/2273733.
- [22] David A Turner (1979): *A new implementation technique for applicative languages*. *Software: Practice and Experience* 9(1), pp. 31–49, doi:10.1002/spe.4380090105.
- [23] Dimitrios Vardoulakis & Olin Shivers (2011): *CFA2: a Context-Free Approach to Control-Flow Analysis*. *Logical Methods in Computer Science* 7(2), doi:10.2168/LMCS-7(2:3)2011.
- [24] Mitchell Wand & Galen B. Williamson (2002): *A Modular, Extensible Proof Method for Small-Step Flow Analyses*. In Daniel Le Métayer, editor: *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002. Proceedings, Lecture Notes in Computer Science 2305*, Springer, pp. 213–227, doi:10.1007/3-540-45927-8_16.